Writing your First Verilog Configuration File
Revision 1.0

John Elliott

October 10, 2012

Abstract

The first file is always the hardest; this is a walkthrough to get LEDs blinking. This howto is designed around the XEM6001, but applicable generally with minor modification.

Tasks

1. Creating a Working Project

The Xilinx suite operates by creating Project files for a specified FPGA and a specific configuration file (CF). For every CF desired, a new project will need to be created.

2. Creating the Files

Make the appropriate files in the project to carry out the FPGA configuration. A .v Verilog file and a .ucf constraints file will be created.

3. Writing the Verilog

The actual Verilog code required to make a simple blinking LED CF.

4. Writing a UCF

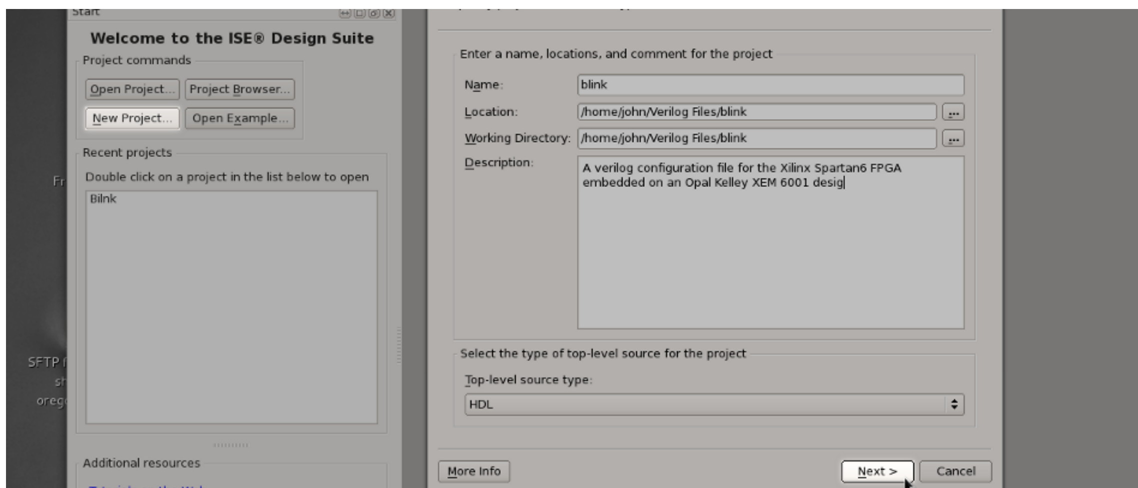
The Universal Constraints File (UCF) will need to be created. Verilog is program code, FPGAs are physical hardware, and UCFs are what tie the two together through Xilinx software.

5. Implementing Code with FPGA

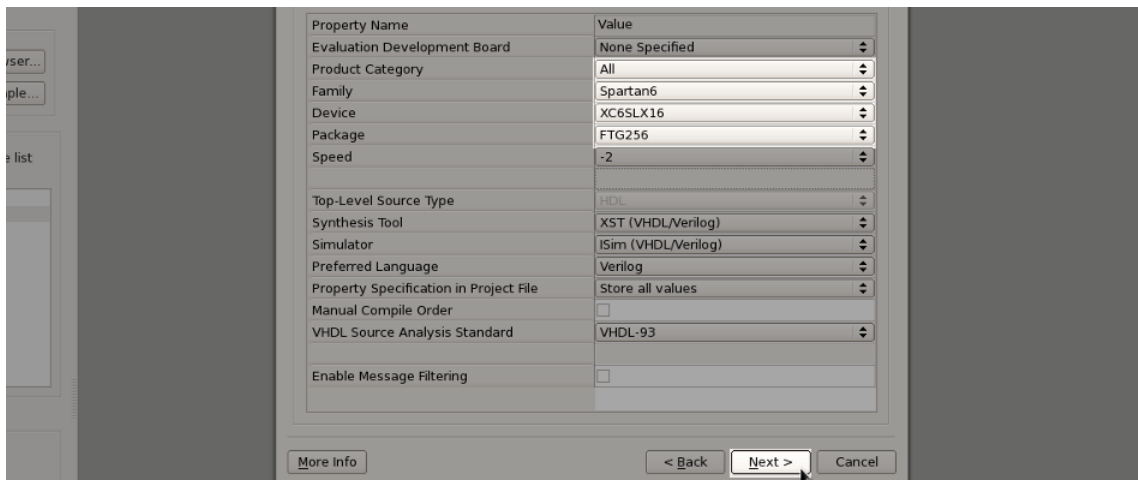
THe neccessary steps required to turn the previous work into a working FPGA that does what the Verilog code outlines.

1. Creating a Working Project

- *Open* Xilinx ISE
- *Click* "New Project" on the left Welcome pane
- Enter in credentials for the new project and *click* "next." Example given below



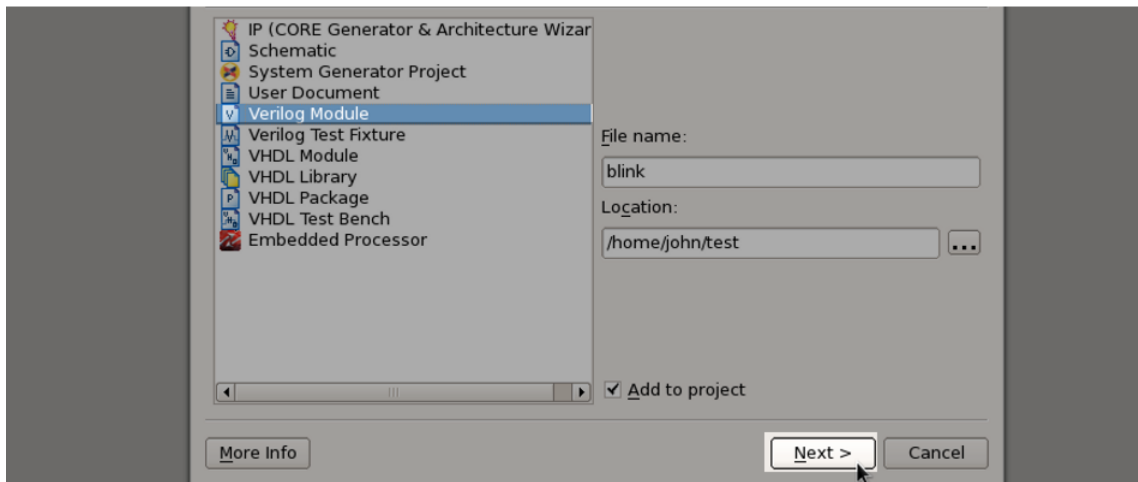
- Configure project for specific Xilinx FPGA (*ex: Spartan6 XC6SLX16-2FTG256*)



- Verify the information presented and *click* "finish"

2. Creating the Files

- *Click* "Project" dropdown menu, then "New Source"
- *Click* "Verilog Module" as the source type and name the file (*ex: blink*)



- *Click* "next" and then "finish" without making any more changes
- Now a Verilog file should be created and open
- Repeat the process, changing the name of the source type to "Implementation Constraints File"

3. Writing the Verilog

*It is now assumed that two files exist in the project: blink.v and blink.ucf. These two names will be henceforth used to refer to the .v file created and the .ucf file created.

- In the ISE, *open* or *select* blink.v so that it can be viewed and edited
- First in a Verilog module is the Input/Output (IO) declarations, which are done inside the parenthesis after the name of the module (after "module blink (" but before ");")
- For this example we will write a Verilog code to blink 8 LEDs as a binary counter
- Declare "input clk" for a system clock, then a comma to separate, and finally "output [7:0] LED" to indicate an 8 bit output named LED (*example below*)

```
20  ////////////////////////////////////////////////////////////////////
21  module blink(
22     input clk,
23     output [7:0] LED
24      );
25
26
27  endmodule
```

- After the IO declarations are made, set the net types for both IOs (the clk is a wire, the 8bit LED array is a register) and additionally anticipate the need for an index (64 bits is a good size for an index) and a register to store the counting number. Since we will be desiring the LEDs to blink on a human time scale, we will also need to generate a synthetic clock that is very slow; lets create a register called "slowclk" to store this synthetic clocks values (0 or 1).

```
25
26    //initializations
27    wire            clk      ;  //This is hardware wired to a clock
28    reg    [7:0]    LED      ;  //This is going to be ports wired to a registry
29    reg    [7:0]    count    ;  //A traditional Registry
30    reg             slowclk  ;  //The register to be used as a synthetic clock
31    reg    [63:0]   i        ;
32
33
```

- Now set the initial values for each of the registers. Also, create a parameter called delay that will be used to tell the configuration how many clock cycles to delay between counts

```
34    //initial values
35    parameter delay = 1_000_000;//How many clock cycles to wait between counts
36                                //This is a constant parameter and is passed to
37    initial                     //the compiler as nothing more than a constant
38       begin
39       LED    [7:0] =  8'hff ;  //My LEDs are held high on one side, this turns them off
40       counts[7:0] =  8'h00 ;   //Counter starts at zero
41       i           =  64'd0 ;   //Counter starts at zero
42       slowclk     =  0     ;   //initial value for the slow clock is zero
43       end
```

- Lets make the slow clock now. Make a program that always runs such that it increases the index if the index is less than the delay and otherwise flips the synthetic clock from high to low or visa versa

```
46    always @(posedge clk)      //At every positive edge of the system clock, start this
47       begin
48       if (i<delay)            //Check to see if we indexed high enough yet
49          begin
50          i  =  i+1    ;
51          end
52       else
53          begin                //Now having indexed enough, invert the synthetic clock
54          i        =  64'd0 ;
55          slowclk = ~slowclk;
56          end
57       end
```

- Finally let the count register tick up always on the positive edge of the slowclk and then port the count register to the LED register so that it can be viewed. The example below additionally shows how to do this with a shorthand notation

```
59    always @(posedge slowclk)
60       begin
61       if (count<256) count =  count+1   ;
62       else           count =  8'd0      ;
63       end
64
65    always  LED [7:0] = ~ count  [7:0]  ;
66
67 endmodule
```

- The Verilog file is now created

4. Writing the UCF

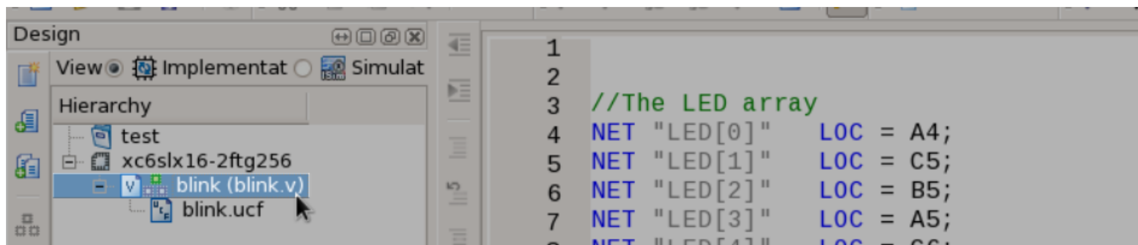
- In the ISE, *open* or *select* blink.ucf so that it can be viewed and edited
- Here the system clock and LED array will need to be tied to the Verilog code. The Verilog file itself is abstract and has no information about the actual chip to be configured; it is the UCF here that will bring the abstract Verilog file into reality
- In a bitwise fashion, connect each Verilog NET to the corresponding FPGA LOC (pin)

```
 3  //The LED array
 4  NET "LED[0]"   LOC = A4;
 5  NET "LED[1]"   LOC = C5;
 6  NET "LED[2]"   LOC = B5;
 7  NET "LED[3]"   LOC = A5;
 8  NET "LED[4]"   LOC = C6;
 9  NET "LED[5]"   LOC = B6;
10  NET "LED[6]"   LOC = A6;
11  NET "LED[7]"   LOC = A7;
12
13  //The system clock
14  NET "clk"      LOC = T8;
```

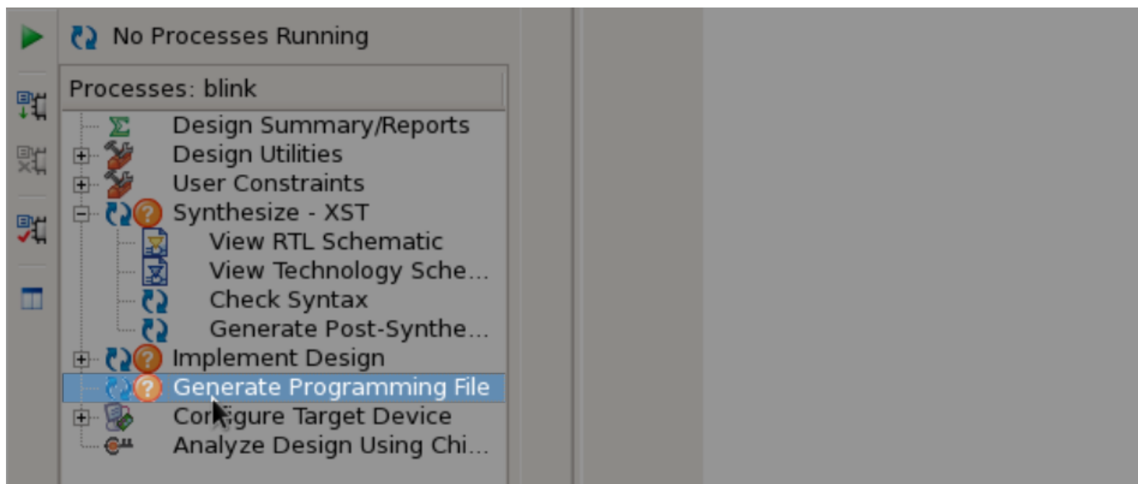
- The UCF is now created and ready

5. Implementing Code with FPGA

- *Select* your top module "blink" from the Implementation Hierarchy window



- *Right – click* "Generate Programming File" from the "Processes: blink" window



- *Click* "Run"

- Watch the "Console" output for a notification that the programming file was successfully generated

- The .bit file is now ready to be loaded. For a walkthrough on how to continue with the XEM6001, view "Opal Kelley Install and Use" manual